INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

TPT-C: A Heuristic-Based Cache to Improve Range Queries over DHTs

Nicolas Hidalgo — Luciana Arantes — Pierre Sens — Xavier Bonnaire

N° 7576

Mars 2011

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7576--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

TPT-C: A Heuristic-Based Cache to Improve Range Queries over DHTs

Nicolas Hidalgo , Luciana Arantes, Pierre Sens * , Xavier Bonnaire †

Thème COM — Systèmes communicants
Équipes-Projets REGAL

Rapport de recherche n° 7576 — Mars 2011 — 23 pages

Abstract: Distributed Hash Tables (DHTs) provide the substrate to build large scale distributed applications over Peer-to-Peer networks. A major limitation of DHTs is that they only support exact-match queries. In order to offer range queries over a DHT it is necessary to build additional indexing structures. Prefix-based indexes, such as Prefix Hash Tree (PHT), are interesting approaches for building distributed indexes on top of DHTs. Nevertheless, the lookup operation of these indexes usually generates a high amount of unnecessary traffic overhead which degrades system performance by increasing response time. In this paper, we propose a novel distributed cache system called *Tabu Prefix Table* (TPT-C), aiming at improving the performance of the Prefix-trees. We have implemented our solution over PHT, and the results confirm that our approach outperforms traditional existing cache solutions for prefix-tree structures.

Key-words: Distributed Cache, Complex Queries, Prefix Hash Trees, Peer-to-Peer, DHT.

* LIP6 - University of Paris 6 - INRIA
† Universidad Técnica Federico Santa María, Valparaíso, Chile

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

TPT-C: Un cache basé sur une heuristique pour amliorer les requêtes par intervalle sur les tables de hachage distribu´es

Résumé : Les tables de hachage distribuées (DHT pour *Distributed Hash Table*) sont un support efficace pour développer des applications à grande échelle sur des réseaux pair-à-pair. La principale limitation des DHT est leur faible pouvoir d'expression. Elles ne supportent que des requêtes pour localiser une information à partir de son identifiant exact. Pour offrir des requêtes par intervalle, il est nécessaire d'ajouter aux DHT des structures d'indexation. Les index par préfix tel que l'arbre de hachage par préfix (PTH pour Prefix Hash Tree) sont une approche intéressante pour construire des index sur des DHT. Cependant, les operations sur ces index génèrent généralement un traffic important qui dégrade les performances du système et augmente les temps de réponse des requêtes. Dans cet article, nous proposons un nouveau cache distribué appelé *Tabu Prefix Table* (TPT-C) pour améliorer les performances des arbres de préfix. Nous avons développé notre solution sur PHT et les résultats confirment que notre approche améliore les solutions de cache proposées précédemment sur PHT.

Mots-clés : Cache distribu, Requtes complexes, Arbre de Hachage par Prefix, Pair--pair, DHT

Contents

1 Introduction

Peer-to-peer (P2P) networks are now widely used to build distributed information systems. In this context, Distributed Hash Tables (DHTs) have shown to be a very efficient solution to implement large scale distributed applications. DHTs are scalable, fault tolerant, and provide load balance. Well-known DHT-based overlays used to build distributed applications are for instance Pastry [16], Chord [19], and CAN [15].

DHTs are distributed systems based on a numerical space in which each node has a unique identifier ($nodeId$) generated using a cryptographic hash function, such as SHA-1. Every data object is associated with a key and is stored in the network by mapping its key to a peer. The $lookup(key)$ operation returns the identifier of the node that stores the key. This operation gives support to hash table operations $put(key, object)$ and $get(key)$ that respectively stores and retrieves an object identifier based on its key. The hashing techniques used in DHTs provide a uniform distribution of the objects within the numerical space, but the lookup operation only supports exact match queries. Therefore, there is no direct method to support efficiently complex queries, like range queries, because uniform hashing destroys data locality [20].

Range queries are required in a wide variety of distributed applications like, music or movie storage, P2P persistent games, scientific computation, data mining, and many types of large scale distributed databases.

A range query retrieves all the objects with values within a given range. For example: *"find all the computers with memory capacity between* $1GB$ *and* $3GB$*"* or *"find all the movies between years* 2000 *and* 2011*"*.

In order to support range queries, many P2P data indexing solutions have been proposed. Among them, those that build an index over the DHT allow to preserve the properties of the underlying overlay, which provides the substrate for building scalable applications. We discuss related work in Section 2.

To improve search over such indexes is a very difficult task: DHT properties should be preserved and there is a trade-off between the latency improvement and the overhead induced by the solution. Techniques such as data replication in upper levels of the index tree are typical solutions. However, they introduce higher index maintenance costs and bottleneck issues.

In this paper we focus on Prefix Trees, and present our solution in the context of Prefix Hash Tree (PHT) proposed in [14]. PHT is one of the most well-known solutions to support range queries over a DHT. It is a trie-like indexing data structure, fully distributed among the peers of the network that can be easily implemented over any DHT. Section 3 details PHT operations and its data structure. We claim that it is possible to significantly improve the performance of the search in Prefix Tree-like structures.

Our main contribution is a new heuristic-based cache for Prefix-Trees called Tabu Prefix Table (TPT-C) based on the Tabu search heuristic [10]. TPT-C uses a shared tabu list to reduce the search space avoiding internal nodes of the trie. These and others features of TPT-C are discussed in Section 4.

With TPT-C, search presents the following improvements:

- Reduction of the message traffic by more than 61% when a linear search is performed.
- Improvement of response latencies up to 80%.
- Reduction of the load of the trie nodes up to 76%.

One of the strong features of our cache technique is that it maintains its performance in dynamic scenarios, since it does not store static-references. Moreover, performance does not depend on data distribution. We discuss in Section 5 the properties and operations of our approach.

Evaluation performance results based on simulation are presented in Section 6. We also compare our results with existing approaches. The last section of the paper is dedicated to the conclusion of our work.

2 Related Work

In order to support range queries which preserve data locality, many P2P data indexing solutions have been proposed in the literature. They can be separated into two classes: *over-DHT* indexing class [14, 20–22], which indexes data over DHTs, and the *overlay-dependent* indexing class which indexes data directly on the overlay [4, 5, 8, 11]. The advantage of the first class is that it is entirely built on top of the DHT interface, and it is thus portable to any DHT. Overlay-dependent indexing solutions adopt either a DHT-free indexing approach which re-designs its own overlay, or a DHT-modification approach where the DHT is modified in order to provide data locality. For instance, in LSH [8], uniform

hash was replaced by the *Locality Sensitive Hash*. However, any indexing data structure solution produces overhead since it generates extra traffic due to the data index structure itself. Another point is that these data structures do not always tolerate churn.

We focus on *Prefix Hash Tree* (PHT) [14] since it is a well-known example of *over-DHT* indexing solution and has been widely exploited [3, 6, 13, 17, 18].

To improve the PHT search process, Chawathe et al. have introduced in [6] a client-side cache system which stores information about leaf nodes. When a new lookup takes place, the client node cache is checked. If it contains the leaf node that keeps the searched data, a *get* operation is performed which aims at verifying that the tree structure has not changed and the node is still a leaf. Therefore, in case of a cache hit, the leaf node is located by a single *DHT-lookup*. Nevertheless, if the trie structure changes or the cache entry is out of date, the traditional search of PHT is performed.

This cache approach improves the PHT performance in static environments, which is however quite unrealistic since P2P systems are essentially dynamic environments. In this case, static-references, such as IP addresses, may become useless since the cache must revert to traditional search methods which generate higher latencies and message overhead. In Section 6 we will show that our approach outperforms this cache solution.

3 PHT: Prefix Hash Tree

The Prefix Hash Tree (PHT) [14] is a trie-like indexing data structure over DHT-based P2P networks. It supports range queries by simply exploiting $get(key)$, and $put(key, object)$ DHT operations.

PHT is an interesting solution to support range queries over DHTs. The indexes build over a DHT preserve the good properties of the latter such as scalability, fault resilience, and load balance. In addition, compared to tree-linked solutions as B-trees, PHT requires only $\log(D)$ lookups, were D is the number of bits of the key, while a B-tree with O objects requires $\log(O)$ lookups. Usually the parameter D is typically 80 bit long. This size can be greater if more attributes are indexed. On the other hand, the size of the parameter O can grow up to millions of objects.

In terms of load balance, lookup operations in tree-based solutions usually start from the root, creating potential bottlenecks and a single point of failure. Note that if a node fails in this structure, the whole sub-tree rooted at the failed node will be lost. On the other hand, PHT lookup operation can "jump" to any node. i.e., it does not require a top down traversal of the trie. PHT does not prevent data loss itself, however, replication provided by the DHT can.

Data Structure

The PHT structure is a binary trie built over the data set where the left branch of a node is labeled 0 and the right branch is labeled 1. Each node n of the trie is identified with a chain of D bits or a prefix produced by the concatenation of the labels of all branches in the path from the root to n. Under the assumption that all objects can be represented by a binary key k, PHT builds a prefix tree

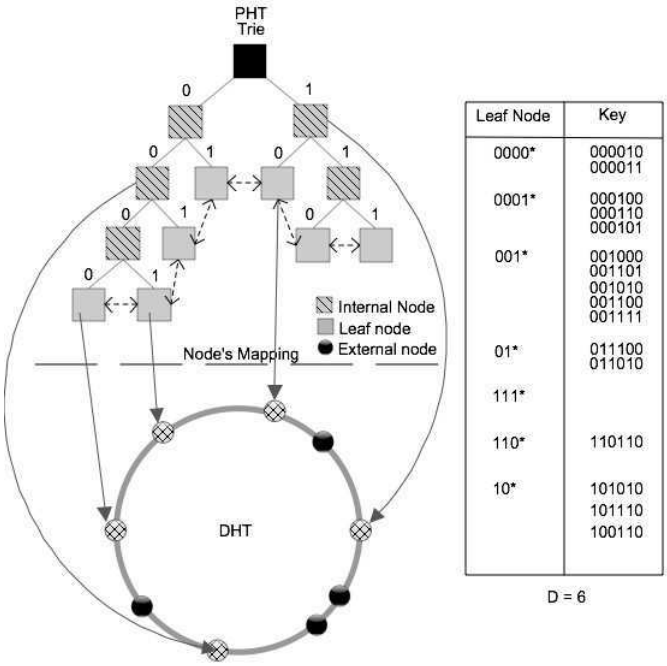


Figure 1: PHT structure: Mapping trie nodes over the DHT.

in which objects are stored at leaf nodes. Hence, an object with key k is stored at a leaf node with a label that is a prefix of K.

The trie structure of PHT is completely distributed among the peers in the network. This is achieved by *hashing* the prefix labels of the PHT nodes over the underlying DHT identifier space. As a consequence, each node of the trie will have an assigned node in the DHT. Thus, each PHT node in the trie corresponds to a node in the DHT. Fig. 1 illustrates an example of mapping. The hash-based assignment allows any object to be located simply by *DHT-lookup* operations. Such an approach enables to run PHT over any DHT-based P2P overlay and therefore the properties of the latter are preserved.

In PHT there are three types of peers: *leaf*, *internal*, and *external*. The first two belong to the PHT trie-structure, while the last one does not, however it participates to the DHT overlay.

In order to improve the performance of range queries, PHT maintains a double list which links all leaf nodes (*Threaded leaves*), as shown in Fig. 1 with dashed lines.

Search: Lookup Operation

Considering a key k with a length of D bits, the *PHT-lookup* returns a unique leaf node $leaf(k)$, whose label is a prefix of k, i.e., the node which stores k. Since there are $D+1$ different prefixes of k, there are $D+1$ potential candidates [14].

Two different search methods can be used to locate a leaf node: *linear search* and *binary search*.

In the *linear search* method, the *PHT-lookup* starts the search by invoking the *DHT-lookup* operation with the shortest prefix of k. Then, if a leaf node is not reached, a new *DHT-lookup* operation with a one-bit longer prefix is performed in order to locate the leaf node. In the worst case, the linear search executes $D + 1$ *DHT-lookups* to reach the leaf node. Note that the linear search can also be performed in *parallel*, i.e., $D + 1$ *DHT-lookups* are executed in parallel, one for each of the $D+1$ prefixes. However, even if this approach provides low-latency lookups, it generates a high overhead in terms of the number of messages.

A more efficient approach in terms of the number of messages is to perform a *binary search*: Binary search is a half-interval process that starts by querying a middle prefix of D. If the prefix corresponds to an internal node of the PHT, the search discards the lower half of the interval and continues querying a new middle prefix of the remaining interval. If the prefix corresponds to an external node, the search discards the upper half of the interval. This search method produces a number of *DHT-lookups* in the order of $\log(D)$. The drawback of binary search is that if an internal node fails or leaves the system, it might be impossible to locate a given leaf node.

Insert/Delete Operations

The insertion of a new object firstly requires to locate the corresponding leaf node. To this end, a *PHT-lookup(key)* must be performed. In order to distribute the load among the nodes, the number of keys stored at a leaf node cannot exceed B. When this B threshold is reached, the leaf node performs a split operation which will produce two children nodes. The keys are then redistributed among these two nodes and the parent node becomes an internal node. It is worth reminding that PHT only stores objects at leaf nodes.

Similarly to the insert operation, the removal of an object firstly requires to locate the corresponding leaf node by performing a *PHT-lookup(key)*. Removal of an object causes the join of two sibling leaf nodes into a single parent node [6].

3.1 Range Queries

Given two keys L and H with $H \geq L$, a range query over PHT will return all the keys contained in the range $[L, H]$. To perform the search, the lower bound is used to find the first leaf node that stores the keys. Once this node is reached, the bidirectional links over the leaves are used to collect all the keys until the leaf node that stores the upper bound is reached.

4 TPT-C: Tabu Prefix Table Cache

In this section, we describe our proposal of a new cache technique for indexing Prefix Trees. This method provides a very efficient persistent shortcut mechanism to avoid re-visiting internal nodes of the tree. The goal of our solution is the improvement of the search operation performance. We develop our solution over PHT, but other Prefix Tree solutions like [1] can also exploit our approach.

Tabu Prefixes Table Cache or TPT-C is inspired by the *Tabu Search* optimization method [10]. This heuristic is used to solve combinatorial optimization

problems, such as the traveling salesman problem (TSP). Tabu search maintains a *Tabu list* which stores the potential solutions already used and thus avoid revisiting the same solution. The principle of *Tabu Search* is to reduce the search space in order to speedup the process of finding an optimal solution. Like *Tabu Search*, the idea of TPT-C is to reduce the search space in the trie aiming at improving latency and reducing message traffic overhead as well as the load of internal levels of the trie.

TPT-C is a flexible and distributed cache solution which provides knowledge about the trie shape. It stores prefix labels of nodes already visited in previous searches that lead to internal nodes. During the search, TPT-C exploits cached information of every visited node in a distributed manner. Furthermore, since TPT-C stores information about internal nodes, and the trie structure is independent of the DHT, churn hardly affects the performance of TPT-C.

It is worth pointing out that TPT-C can be used independently of the search method. However, it has a major impact when used in conjunction with linear search (see Section 6).

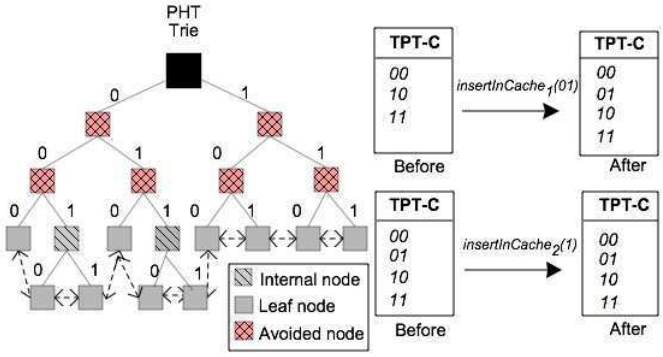


Figure 2: TPT-C insert operation

4.1 Insert in Cache

Each entry in TPT-C stores a prefix label p, which corresponds to an internal node of the PHT. The number of entries in TPT-C per node is limited to ϵ entries. When a node wants to insert a prefix in its local cache, it calls the *Insert in Cache* function. The prefix will be inserted in the cache if it is not already stored and it is not *redundant*.

The information about the prefix related to an internal node is *redundant* if a children is already present in TPT-C. The idea is to preserve, whenever possible, the longest prefixes in the table and remove the unnecessary entries. The insertion is presented in Algorithm 1.

Fig. 2 shows two examples of an entry insertion in TPT-C. When the $insertInCache_1(01)$ is performed, TPT-C-Insert inserts the entry because it is not present or redundant in the table. In operation $insertInCache_2(1)$, the entry with label 1 is not inserted because it is *redundant* in TPT-C. When the number of entries reaches ϵ, traditional strategies are used for entry replacement: *First in First Out* (FIFO), *Least Recently Used* (LRU) and *Least Frequently Used* (LFU).

Algorithm 1: Insert in Cache (insertInCache)

```
input : Entry e
strategy ∈ {FIFO, LRU, LFU};
Boolean insert_flag = true;
if TPT-C.size = ϵ then
    removeEntry (strategy);
foreach Entry f in TPT-C do
    /* check if e is not redundant */
    if commonPrefixLength (e, f) ≥ e.size then
        insert_flag = false;
if insert_flag then
    insert (e);
    foreach Entry f in TPT-C do
        /* remove redundant entries after the insert */
        if commonPrefixLength (e, f) = f.size then
            removeEntry (f);
```

4.2 TPT-C Search Algorithm

In traditional PHT algorithms when a PHT-lookup of a key k is performed, the querying node Q_n iteratively generates a *DHT-lookup* with a different prefix of k ($P_i(k)$ prefix of length i) according to the selected search method (linear or binary). In TPT-C, when Q_n starts the PHT-lookup, it first checks its own cache in order to find an entry which can reduce the search space.

The TPT-C check is presented in Algorithm 2. Check in cache iteratively searches the table in order to find a useful entry. The check consists in finding the entry which has the *greatest common prefix length* (*gcp*) with k. In order to go forward in the search, the entry found must be longer than the current prefix length of the *DHT-lookup*. If an entry is found, there is a cache hit, and the entry can be used to continue the search. If no entry is found, the function returns *null*.

Algorithm 3 shows a new query issued by a node Q_n of a key k. When a local cache hit takes place, the node performs a new *DHT-lookup* starting from one step further than the found prefix of the cache entry. If no entry is found, the lookup operation is performed as usual. Note that the next prefix used in the lookup (`nextPrefix`) depends on the search method (linear or binary).

A *DHT-lookup* is sent from Q_n to a remote node R_n which is the destination node of the searched prefix. Algorithm 4 presents the reception of a lookup operation at node R_n. If R_n is a leaf node it directly sends a reply message to Q_n including its type. Otherwise, if R_n is an internal or an external node, it checks its local cache to find if there is an entry which can reduce the search space. If an entry e is found, R_n adds e to the reply message to Q_n.

Algorithm 5 presents the reception of a lookup-reply message by node Q_n. Q_n executes one of the following actions: If the reply is from a leaf node, Q_n knows that the search has ended and gets the searched data by contacting node R_n. If the lookup-reply message includes a cache entry, the latter is inserted in the local cache and the lookup continues one step further from the cache entry

Algorithm 2: Check in Cache (checkInCache)

input : Key k, int $prefixLength$
output: Entry hit

int $gcp = prefixLength$;
/* `gcp denotes the greatest common prefix length` */
Entry hit = ¡null¿;
foreach *Entry e in TPT-C* **do**
  if `commonPrefixLength` (e, k) ¿ gcp **then**
    gcp = `commonPrefixLength` (e, k);
    $hit = P_{gcp}(k)$;

Algorithm 3: New Query at Q_n

input : Key k
output: Message m

$prefixLength = 0$;
Entry e = `checkInCache` (k, $prefixLength$);
if e = *¡null¿* **then**
  m = `lookup` (`nextPrefix` (k, $prefixLength$));
else
  m = `lookup` (`nextPrefix` (k, $e.prefixLength$));

Algorithm 4: Lookup Processing at R_n

input : Key k, int $prefixLength$
output: Message m

if $myNodeType = 'leaf'$ **then**
  `lookupReply` ('leaf', null);
else
  Entry e = `checkInCache` (k, $prefixLength$);
  if e != *¡null¿* **then**
    m =`lookupReply` ($myNodeType$, e);
  else
    m = `lookupReply` ($myNodeType$, null);

found. Otherwise, the search continues as usual. If the reply is from an internal node, the searched prefix is also stored in the local cache.

Algorithm 5: Lookup-Reply Processing at Q_n

```
input  : Key k, int prefixLength, NodeType type, CacheEntry c
Output: Message m
if type = 'leaf' then
    m = getData ();
else
    if c != null then
        insertInCache (c);
        m = lookup (nextPrefix (k, c.prefixLength));
    else
        m = lookup (nextPrefix (k, prefixLength));
        if type = 'internal' then
            insertInCache (P_prefixLength(k));
            /* P_i(k) denotes the prefix label of length i of the
            key k */
```

5 Discussion

In this section we analyze some important aspects of using TPT-C, the search methods, and dataset distributions.

5.1 Using TPT-C

Traditional cache techniques of indexes store information about the nodes that keep data. In the case of PHT, the data is stored only at the leaf nodes. By exploiting this information, a node can directly contact the leaf node when it needs to perform a lookup operation. However, this cache information usually consists of a key and an IP address, which is a static reference to the node. Hence, if the node leaves the network, such an entry in cache will be useless for future queries. TPT-C does not use static references. Instead, it exploits only logical information about the tree. By gathering information from internal nodes, TPT-C can reduce the search space over the tree even when nodes join or leave the DHT. Therefore, it tolerates P2P churn.

One of the properties of binary tries is that they have a similar number of leaf nodes and internal nodes [20]. Thus, the potential information to store in cache is the same if we store internal nodes as TPT-C or leaf nodes as a traditional cache. When a query is performed in PHT many internal nodes and only one leaf node are visited. This is the reason why TPT-C gathers cache information faster than a traditional cache does.

Another benefit of storing information about internal nodes is that it improves the search for all the queries which have the sub-tree rooted at the internal node as destination. Hence, compared to a traditional cache, a greater number of queries exploits the information stored in TPT-C.

As we mentioned previously, TPT-C can be used independently of the search method. However, it presents a major performance impact when used in conjunction with linear search. When binary search is used, TPT-C search space might be slightly reduced since only the lower bound of the binary search can be optimized. The upper bound never changes because it is impossible to deduce the depth of the current branch in which the search takes place.

Maintaining load balance among the nodes of the trie is a major concern. When linear search is performed, the nodes closer to the root have to deal with a high number of queries compared to nodes of the lower levels of the trie, i.e., far from the root. When TPT-C is used, accesses to internal nodes are reduced. TPT-C diverts queries to the lower levels of the trie where queries are spread over a greater number of nodes compared to the higher levels. Therefore, TPT-C provides better load balance.

Since storage requirement (σ) of TPT-C is low, it is possible to maintain a high number of entries (ϵ) in cache to improve search performance. Each cache entry of TPT-C consists of a key prefix which in the worst case is the number of bits of the indexed space (D bits). In [6] an 80 bit key was used for a real application. In TPT-C, for the same key size, 100 entries would require only $100 \times 80 = 8000$ bits, approximately 1Kb.

In the case of multi-attributes range queries, key size would increase linearly with the number of attributes (α). For example, if we consider an 80 bit key and a space linearization technique like *Space Filling Curves* (SFC), a α-attribute resource will be represented by an $\alpha \times 80$ bits key. In consequence, the maximum storage requirement for TPT-C can be estimated as $\sigma_{max} = D * \epsilon * \alpha$.

5.2 Search Methods

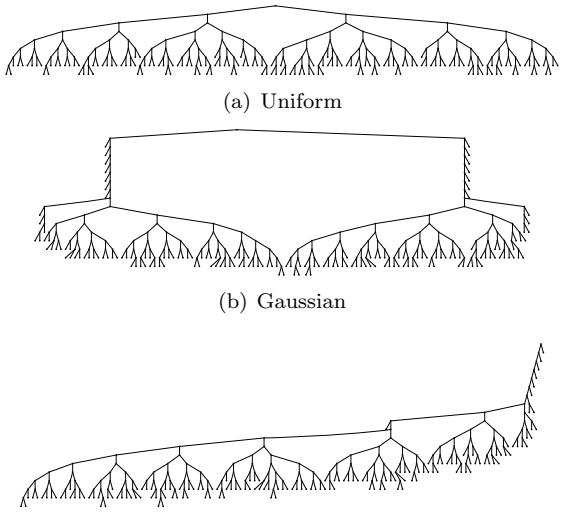
As mentioned in Section 3, a query in PHT can use *Binary*, *Linear* or *Linear parallel search* methods.

Binary search is an effective search method, however it presents some drawbacks on dynamic scenarios. In PHT, *binary search* can fail locating leaf nodes. Both the join and leave of nodes, and PHT insert and delete operations produce trie-shape modifications. Thus, wrong answers about the nodes state in the trie can be propagated compromising the search. Since binary search does not go backwards, the search might finish without result. Such a behavior introduces extra traffic overhead and latency, because a new search should be performed, using the same method or the linear search one.

Linear search is not as efficient as binary search in terms of number of lookups messages and latency. Nevertheless, it always returns a leaf node despite the dynamism of the system.

Linear Parallel search is a low-latency solution. When using parallel search all the prefixes are evaluated at the same time. This generates $D + 1$ *DHT-lookup* messages introducing high overhead in the system. The consequences of this high traffic are twofold: it degrades DHT performance and internal nodes in trie are overloaded answering useless queries. Note that linear parallel search cannot exploit any cache technique due to its parallelism. Consequently, searches will present low-latency but high-overhead.

In [6], a real application over PHT is presented. To index data they used 80 bits keys. Considering a system in which $1,000,000$ queries are generated, parallel search will induce a total traffic of $1,000,000 \times (80 + 1) = 81,000,000$



(a) Uniform

(b) Gaussian

(c) Pareto

Figure 3: PHT generated with Uniform, Gaussian and Pareto data distributions.

messages. More than 98% of this traffic is useless. Therefore, due to this huge overhead and inability to exploit cache techniques, we have not considered the parallel search in our evaluation experiments. We argue that it is not applicable to real scenarios.

5.3 Data Distribution

PHT search efficiency is very sensitive to data distributions. Data distribution modifies the trie shape and thus has an impact on lookup performance. For instance, in clustered datasets, data are placed at lower levels of the trie. Therefore, querying such data will require a higher number of *DHT-lookups*. We refer to clustered data as a high density of objects within a given range of keys.

Uniform distributions of keys generate balanced tries in which leaf nodes are located on a similar level. Fig. 3(a) shows an example of such a distribution. All queries need almost the same number of lookup operations to find a leaf node. For example in Fig. 3(a), between 6 and 8 lookups are needed.

Skewed distributions like Gaussian and Pareto generate unbalanced tries. In Gaussian distribution, generated with mean μ and standard deviation σ, the lowest levels of the trie store the keys which are closer to μ. Note that almost 68% of the keys are concentrated in the range $[\mu + \sigma, \mu - \sigma]$. Fig. 3(b) shows a trie generated with a Gaussian distribution.

Pareto distribution is a typical long-tail distribution. Where a huge amount of data is concentrated within a small range of keys. In this case, the trie shape is fully unbalanced. Fig. 3(c) presents a trie generated with Pareto distribution. In order to locate the clustered data, a higher number of *DHT-lookups* are necessary compared to sparse data.

In real applications, Uniform distributions are very rare, however Pareto and Gaussian are highly common [2, 9].

6 Performance Evaluation

This section provides a performance evaluation of TPT-C by comparing our approach with the traditional PHT and the cache system presented in [6] (see Section 2), denoted *Cache* in this section. Our simulations were conducted on top of the Peersim simulator [12]. Our aim is to show the effectiveness of our cache technique in terms of network traffic and query latency. Additional studies were also performed to evaluate the influence of data distribution in our approach. Finally, we have evaluated the impact of dynamic scenarios on TPT-C, and present the analysis of both the number of entries and replacement strategies.

6.1 Experiments Setup

In order to conduct the experiments, we have built an event-like simulation based on an already existing PeerSim module which simulates Pastry layer [7]. We have implemented a PHT layer on top of Pastry with TPT-C and *Cache*.

We have considered a network of $10,000$ peers and that a PHT node stores at most 100 objects ($B = 100$). $100,000$ objects have been inserted in the network to generate the trie. Keys related to data objets were generated using three different distributions: *Uniform*, *Gaussian* and *Pareto*. The probability density function of *Pareto* is defined as: $\frac{\alpha\beta^{\alpha}}{x^{1+\alpha}}$, where $x \geq \beta$ and, α and β respectively denote the tail length shape and the lower bound. Pareto dataset has a Pareto key distribution with parameters $\alpha = 2$ and $\beta = 1$ [4, 9]. For the Gaussian datasets, keys have a gaussian distribution centered in the range of the dataset.

A total of $2,000,000$ queries were simulated for each experiment, which were also generated using the above three distributions. A snapshot of the system is taken every $100,000$ queries. Both linear and binary searches methods were evaluated.

In our experiments the following aspects were evaluated:

- Query latency and message traffic overhead.
- The load balance of the trie nodes.
- The impact of the number of cache entries.
- The impact of three well-known cache replacement strategies: *First-in First-out* (FIFO), *Least Recently Used* (LRU), and *Least Frequently Used* (LFU).
- The impact of dynamic scenarios.

6.2 Message Traffic Overhead and Query Latency

In order to evaluate the performance of TPT-C compared to the traditional PHT and PHT with *Cache*, we have measured query latency and message traffic overhead in the system. Latency is measured by the number of *DHT-lookups* to reach the searched data. We have also compared the number of cache hits and how both approaches (TPT-C and *Cache*) gather cache information.

The parameters of the experiments are:

- Search method: Linear (L) and Binary (B).
- Dataset distribution: *Uniform*, *Gaussian*, and *Pareto*.
- Cache strategy: LRU.
- Cache size: 100 entries.

Table 1 presents the message traffic when linear search is used. TPT-C strongly reduces the total message traffic. The percentage of reduction was calculated comparing PHT with *Cache* (% *Cache* Gain) and PHT with TPT-C (% TPT-C Gain). Notice that linear search in conjunction with TPT-C induces a reduction of more than 61% of the total message traffic when an Uniform distribution is used. In skewed distributions like Pareto, there is more than 78% of reduction. On the other hand, *Cache* (% *Cache* Gain) reduces the total message traffic in only 5.4% and 5.3% when the dataset distribution is Uniform and Pareto respectively.

Table 1: Message Traffic with Linear Search

	Uniform	Gaussian	Pareto
PHT	41,561,075	57,662,472	77,060,642
Cache	39,318,907	54,718,598	72,994,783
TPT-C	16,117,593	16,700,462	16,452,946
% *Cache* Gain	5.40	5.10	5.30
% TPT-C Gain	61.22	71.04	78.65

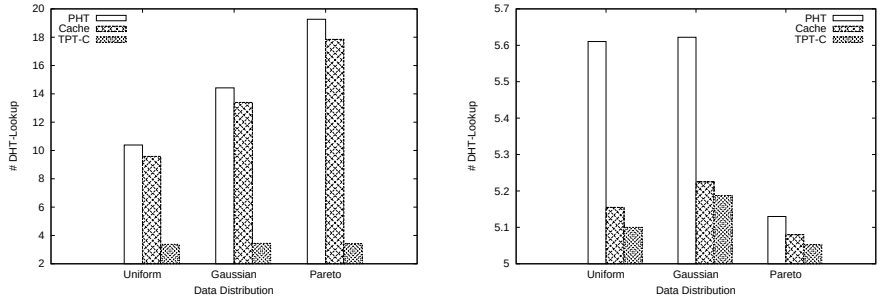
The difference between Uniform and skewed distributions is the height of the trie. In Uniform distribution the trie grows more horizontally while in skewed distributions the growth is vertical.

A second interesting result can be observed in Table 1. Our approach "artificially decreases" the height of the trie by avoiding internal nodes. The practical consequence is that the height of the trie has no longer an impact on the message traffic as it happens with PHT and *Cache*. Our solution thus, produces approximately the same amount of messages independently of the height of the trie.

Table 2: Message Traffic with Binary Search

	Uniform	Gaussian	Pareto
PHT	22,421,522	22,478,237	20,544,039
Cache	21,211,486	21,404,470	20,330,828
TPT-C	21,082,422	20,713,470	20,259,175
% *Cache* Gain	5.40	4.78	1.04
% TPT-C Gain	5.97	7.85	1.39

As we previously mentioned in Section 3, binary search is an efficient search method and the use of a cache like TPT-C does not have a great impact on it.



(a) Latency - Linear (b) Latency - Binary

Figure 4: Query latencies with linear and binary searches.

Table 2 shows the total message traffic when binary search is used. Compared to PHT, the total traffic of Uniform and Pareto distributions are respectively reduced in 5.97% and 1.39%. TPT-C cannot reduce the search space by constraining the upper bound of the binary search because there is not enough information to deduce how deep is the current branch of the trie. For this reason, TPT-C can only reduce search space by constraining the lower bound.

Fig. 4 shows query latencies for PHT, *Cache* and TPT-C. As can be observed, TPT-C effectively reduces the latency when linear search is performed.

The obtained results are promising since TPT-C in conjunction with linear search presents better performance than the binary search with *Cache* or PHT without cache. Furthermore, in the Uniform case, PHT with binary search has an average of 5.61 *DHT-lookups*, while TPT-C with linear search has an average 3.37 *DHT-lookups*, reaching 40% better performance. These results are also valid for the other two distributions evaluated. It is worth emphasizing that the use of linear search with TPT-C is even more attractive than binary search since the former effectively supports churn and the latter does not.

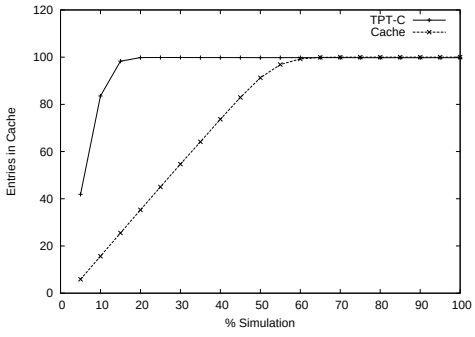


Figure 5: Number of cache entries.

In Section 5, we mentioned that TPT-C gathers cache information faster than classical cache techniques which collect information only about the leaf nodes of the trie. Fig. 5 shows how cache tables are filled with information during simulations for both TPT-C and *Cache* conducted with Pareto distribution and

linear search method. As expected, TPT-C fills its cache table faster than *Cache* does. At each lookup operation, TPT-C can collect a higher amount of information about internal nodes, while *Cache* only gets one cache entry per lookup. TPT-C requires less than 20% of the simulation time to fill its tables, while *Cache* requires 3 times more.

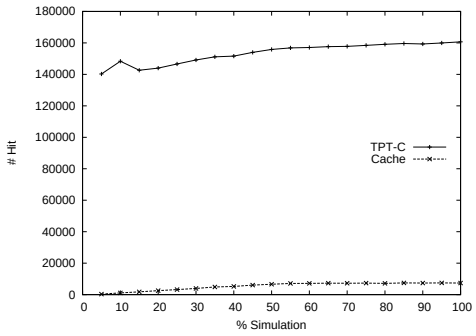


Figure 6: TPT-C and *Cache* hits progress.

Fig. 6 compares the number of cache hits of both *Cache* and TPT-C with the linear search and Pareto distributions. TPT-C largely outperforms *Cache* since with TPT-C a greater number of queries reaches faster a leaf node. Even thought our approach provides more hits, one hit does not lead to a leaf node directly as *Cache* does. On the other hand, in dynamic scenarios, a hit of *Cache* may fail to get the data as previously discussed.

6.3 Data Distribution

We have evaluated the impact of different dataset distributions in the performance of TPT-C with linear search: *Uniform*, *Gaussian* and *Pareto* distributions. Uniform distribution induces balanced tries, while Gaussian and Pareto distribution induce unbalanced ones.

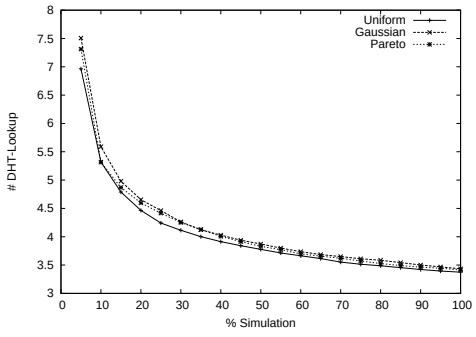
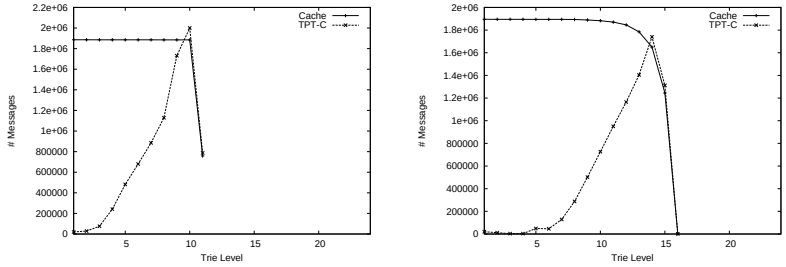


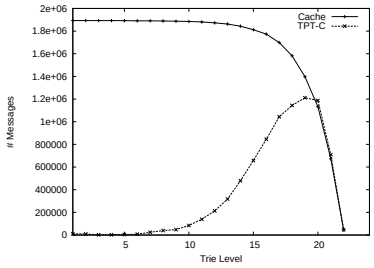
Figure 7: Impact of different key distributions over TPT-C search.

Fig. 4 shows that in the case of linear search, TPT-C efficiently reduces the query latencies independently of data distribution. In the case of binary



(a) Uniform

(b) Gaussian



(c) Pareto

Figure 8: Trie nodes load for TPT-C and *Cache* with different dataset distributions.

search, Fig. 4 shows that our approach performs similarly to PHT and *Cache* independently of data distribution.

Fig. 7 shows how the number of *DHT-lookups* evolve during the simulation. We can observe that there is almost no difference between the three distributions. Query latency quickly decreases while cache tables are being filled. After 20% of simulation time, tables are full and the query latencies decreases slowly.

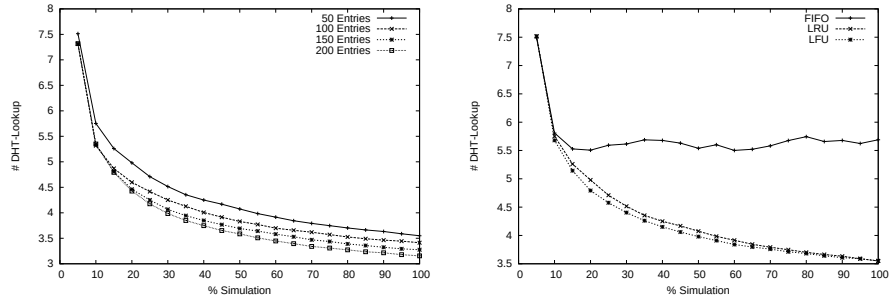
6.4 Load Balance

One of the typical problems of tree-like data structures is that a top-down traversal search is usually used. Upper levels nodes, close to the root, have to deal with a higher traffic than lower levels nodes. In case of PHT linear search, such traffic does not return any result because only leaf nodes store data information. Linear search works iteratively by trying at each step a greater prefix until reaching a leaf node. This search overloads the upper level internal nodes, that can thus become a bottleneck. Note that in the upper levels of the trie the total traffic is only spread over few nodes.

The goal of this experiment is to show that TPT-C improves load balance of linear search since it diverts queries to the lower levels of the trie, closer to the leaf nodes. Therefore, traffic is distributed among a higher number of nodes when compared to upper levels.

Experiments parameters are the same as the ones of Section 6.2.

The load of nodes in the trie structure is presented in Fig. 8, where the horizontal axis represents the levels of the trie (root = level 0; the highest level).



(a) Size (b) Strategies

Figure 9: Cache size and replacement strategies.

Using *Cache*, linear search entails the same load to all internal levels of the trie. Since the number of nodes in higher levels is smaller than in lower levels, their load can become critical. In the case of the Uniform distribution of the Fig. 8(a), the load that is distributed among 2^8 nodes is the same that the one distributed among the 2^1 nodes of the first level.

TPT-C effectively reduces load on the highest levels by avoiding DHT-Lookups over these nodes, querying directly lower levels of the trie.

Note that in Fig. 8, the lowest levels present less traffic. In case of Uniform distribution, presented in Fig. 8(a), the lowest levels have more traffic compared to Fig. 8(b) and 8(c) because a higher number of leaf nodes are present in such levels. In skewed distributions, there are leaf nodes in almost all the levels of the trie and therefore less traffic is produced in the lowest levels.

6.5 Cache Size and Replacement Strategies

Due to the limited storage capabilities of nodes, it is not realistic to consider that node cache tables have unlimited size. However, in Section 5 we have shown that cache storage requirements for TPT-C are low. The number of cache entries have a direct impact in the TPT-C performance, because TPT-C uses cache information to avoid internal nodes and thus to improve search performance.

The goal of this experiment is to measure the impact of the number of entries in cache table on performance of TPT-C.

The parameters of the experiments are:

- Search method: Linear.
- Dataset distribution: *Pareto*.
- Cache strategy: LRU.
- Cache sizes: 50, 100, 150, and 200 entries.

The results of such experiments are presented in Fig. 9(a). As we can observe, cache size has a direct impact on the performance of TPT-C. With 50 entries latency is degraded up to 19% when compared to 100 entries. On the other hand, when 150 entries are cached, latency is improved in 14% compared

to 100 entries. A cache with 200 entries outperforms in 11% a cache with 150 entries. Note that the efficiency of the cache increases with the cache size.

Surely, there is a point where no performance improvement is achieved, when the number of cache entries increases. If the size of the cache was unlimited, TPT-C would have knowledge about all the internal structure of the trie allowing to reach leaf nodes with a single *DHT-lookup*.

As expected, TPT-C table size is an important parameter for TPT-C performance, but its performance does not increase linearly with the cache table size.

As cache size is limited, cache replacement strategies to manage the removal of entries must be used. We have evaluated the impact of using FIFO, LRU, and LFU in TPT-C. A small cache size of 50 entries was chosen aiming at inducing a high number of entries replacement in cache.

The parameters of the experiments are:

- Search method: Linear.
- Dataset distribution: *Pareto*.
- Cache strategies: FIFO, LRU, and LFU.
- Cache size: 50.

Fig. 9(b) shows the results of these experiments. LRU and LFU strategies present almost the same performance during the experiments. As expected, FIFO strategy does not perform well because it removes entries from cache table which can be useful for future queries. In consequence, we can state that LRU and LFU are both good cache replacement strategies to be used with TPT-C.

6.6 Dynamic Scenarios

Since P2P systems are highly dynamic, in the following experiments we have measured the performance of *Cache* and TPT-C when nodes join and leave the DHT randomly under churn conditions. We have evaluated query latencies using the following parameters:

- Search method: Linear.
- Dataset distribution: *Pareto*.
- Cache strategy: LRU.
- Cache size: 100 entries.
- Churn rate: 10% of nodes leave or join the DHT randomly in every simulation window.

Simulation is divided of windows of 100,000 queries. Churn is introduced during a full window followed of another without churn. A 10% of churn at each simulation window was enough to prove that *Cache* performance is highly impacted by dynamic scenarios.

Fig. 10 presents how *Cache* and TPT-C performance is degraded in a dynamic scenario. Performance degradation percentage is calculated between performance improvement introduced by each technique with and without churn.

TPT-C outperforms *Cache* in dynamic scenarios. As can be observed, TPT-C is slightly affected by churn degrading its performance in a 3.6%, instead *Cache* is highly impacted, performance was degraded in 70%. Since TPT-C exploits the logical information of the trie to implement its cache, there is only a small difference between its performance with and without churn. This difference is due to new nodes that join the system with empty cache tables. On the other hand, *Cache* is clearly more affected by churn. When nodes leave the network, static references as IP addresses are useless and thus *Cache* cannot provide any improvement to the search method.

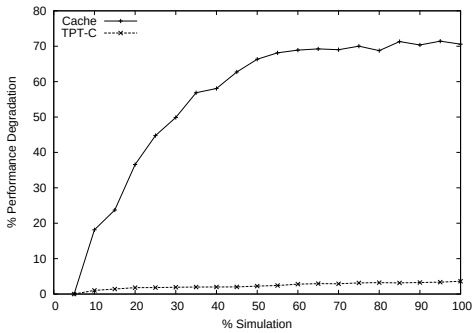


Figure 10: Performance of *Cache* and TPT-C on dynamic scenarios

7 Conclusion

We have proposed in this paper a new cache for distributed Prefix Trees called Tabu Prefix Table Cache (TPT-C), aiming at improving the performance of PHT lookup operations. TPT-C is a solution based on the *Tabu Search* heuristic technique. By exploiting the Tabu principle, TPT-C reduces the search space and therefore both message traffic and query latencies are improved since the number of DHT-lookups performed over internal nodes are reduced. Moreover, internal nodes of the trie are released from unnecessary traffic enabling better load balance among the nodes.

TPT-C artificially decreases the height of the trie by avoiding the internal nodes. The practical consequence of this reduction is that the height of the trie has no longer any impact on the traffic messages as it happens in PHT and *Cache*. Thus, TPT-C is efficient independently of data distribution. Results confirm that our solution produces approximately the same amount of messages independently of the height of the trie outperforming traditional cache solutions.

Another interesting feature of TPT-C is that it tolerates churn since it exploits logical information instead of static references, such as node IP addresses. We should point out that TPT-C gathers cache information faster than other techniques which allows to minimize the impact of information loss when nodes join or leave the network.

Evaluation results show that TPT-C with linear search outperforms by 61% PHT and *Cache* approaches. With skewed distributions like Pareto, the improvement goes up to 78%. In this context, load balance is also highly improved.

References

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3):29–33, 2003.

[2] A. Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. In *Peer-to-Peer Computing, 2002. (P2P 2002). Proceedings. Second International Conference on*, pages 33 – 40, 2002.

[3] E. Ansari, G. H. Dastghaibifard, M. Keshtkaran, and H. Kaabi. Distributed frequent itemset mining using trie data structure, 2008.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[5] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: a multi-attribute addressable network for grid information services. In *Grid Computing, 2003. Proceedings. Fourth Int. Workshop on*, pages 184–191, Nov. 2003.

[6] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered dht applications. In *SIGCOMM '05: Proceedings of the 2005 Conf. on Applications, technologies, architectures, and protocols for computer communications*, pages 97–108, New York, NY, USA, 2005. ACM.

[7] M. Cortella and E. Bisoffi. Peersim pastry: An implementation of the pastry protocol for peersim.

[8] M. Datar and P. Indyk. Locality-sensitive hashing scheme based on p-stable distributions. In *In SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM Press, 2004.

[9] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. In *In P2P'05: Proceedings of the 5th international conference on peer-to-peer computing*, page 5, 2005.

[10] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[11] H. V. Jagadish, B. Chin Ooi, and Q. Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *In VLDB*, pages 661–672, 2005.

[12] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[13] J. Liang and K. Nahrstedt. Randpeer: Membership management for qos sensitive peer-to-peer applications. In *INFOCOM*, 2006.

[14] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *In Proceedings of ACM PODC, St. Johns, Canada, July 2004*, 2004.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[16] A. Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, June 25 2001.

[17] D. Smith, N-F. Tzeng, and M. M. Ghantous. Fasred: Fast and scalable resource discovery in support of multiple resource range requirements for computational grids. In *NCA '08: Proceedings of the 2008 Seventh IEEE Int. Symposium on Network Computing and Applications*, pages 45–51, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Fabian Stäber, Gerald Kunzmann, and Jörg Müller. Extended prefix hash trees for a distributed phone book application. *International Journal of Grid and High Performance Computing (IJGHPC)*, Vol. 1, No. 4, pp. 57-69 pp., 10 2009.

[19] I. Stoica, R. Morris, D. Liben-nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications, July 29 2001.

[20] Yuzhe Tang and Shuigeng Zhou. Lht: A low-maintenance indexing scheme over dhts. In *IEEE Int. Conf. on Distributed Computing Systems (ICDCS 2008)*, pages 141–151, 2008.

[21] X. Wei and K. Sezaki. Dhr-trees: A distributed multidimensional indexing structure for p2p systems. In *ISPDC '06: Proceedings of the Proceedings of The Fifth Int. Symposium on Parallel and Distributed Computing*, pages 281–290, Washington, DC, USA, 2006. IEEE Computer Society.

[22] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over dht. In *In Electronic publications of the 5th Int. Workshop on Peer-to-Peer Systems (IPTPS '06)*, 2006.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399